CME 194 Introduction to MPI

Communi Opera

http://cme194.stanford.edu

Recap

- **Last class:** Point to Point Communication
  - Send/Receive allows us to **correctly** implement **any** parallel algorithm.
  - However **efficient** algorithms may be **difficult**
- **This class:** Collective Operations
  - Makes writing **efficient** distributed memory easier

Example: Prefix Sum

Given an array of **n** numbers, produce

$$\sum_{i=0}^{k} \text{ for each } k \in 0, \ldots, n$$

Prefix Sum

Input:

8	1	13	6	4	2	1	7	5	3

Correct Answer:

8	9	22	28	32	34	35	42	47	50

Serial Algorithm:

```
int main( int argc*, char * argv){
        const int A[10] = {8,1,13,16,4,2,1,7,5,3};
        const int result[10] = {0,0,0,0,0,0,0,0,0,0};

        result[0] = A[0];
        for (std::size_t i = 1; i < 10; ++i){
                result[i] = result[i-1] + A[i];
        }
        return 1;
}
```

Prefix Sum: Parallel Algorithm

Step 1: Local Approximation

8	9	13	19	4	6	1	8	5	8

Prefix Sum: Parallel Algorithm

Step 1: Local Approximation

8	9	13	19	4	6	1	8	5	8

Correct! | Off by 9 | Off by 28 | Off by 34 | Off by 39

Step 2: Recursion

9	19	6	8	8

$\Longrightarrow$

0	9	28	34	39

Fixing the offsets is another prefix sum problem!

Prefix Sum: Parallel Algorithm

Warning: Do **NOT** Implement this procedure. Ever. **Instead:**

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
      MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Scan applies op on sendbuf for ranks 0,…,i
versus
Exscan does Op for ranks 0, .. ,(i-1)

```
int MPI_Exscan(void *sendbuf, void *recvbuf, int count,
      MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Prefix Sum: Parallel Algorithm

Psuedocode:

```
#include "mpi.h"
#include <vector>
#include <iostream>
#include <random>

typedef std::mt19937 rng_type;
std::uniform_int_distribution<rng_type::result_type> udist;

int main( int argc*, char * argv){
        MPI_Init( &argc, &argv);
        if( argc != 2){
           std::cerr << "Usage: " << argv[ 0] << " n" << std::endl;
        }
        int n = atoi( argv[ 1]);
        rng_type rng;
        std::vector< int> A( n, 0);
        for( auto  & i: A){ i = udist( rng); }
        std::vector< int> result( A);
        for (auto i = ++result.begin(); i != result.end(); ++i){ *i += *(i-1); }
        int offset=0;
        MPI_Exscan( &result.back(), &offset, 1,
                    MPI_INT, MPI_SUM, MPI_COMM_WORLD);
        if( offset != 0){
          for( auto i = result.begin(); i != result.end(); ++i){ *i += offset; }
        }
        return 1;
}
```

Correct solution in

$$O(\frac{n}{p} + p\log(p))$$

MPI Operations

- **MPI_MAX** return the maximum
- **MPI_MIN** return the minumum
- **MPI_SUM** return the sum
- **MPI_PROD** return the product
- **MPI_LAND** return the logical and
- **MPI_BAND** return the bitwise and
- **MPI_LOR** return the logical or
- **MPI_BOR** return the bitwise of
- **MPI_LXOR** return the logical exclusive or
- **MPI_BXOR** return the bitwise exclusive or
- **MPI_MINLOC** return the minimum and the location
- **MPI_MAXLOC** return the maximum and the location

Roll your own:

```
MPI_Op_create( MPI_User_function* fp,
               int commute,
               MPI_Op *op);
```

```
void MPI_User_function( void * invec, void * inoutvec, int * len, MPI_Datatype *datatype)
```

MPI Operations

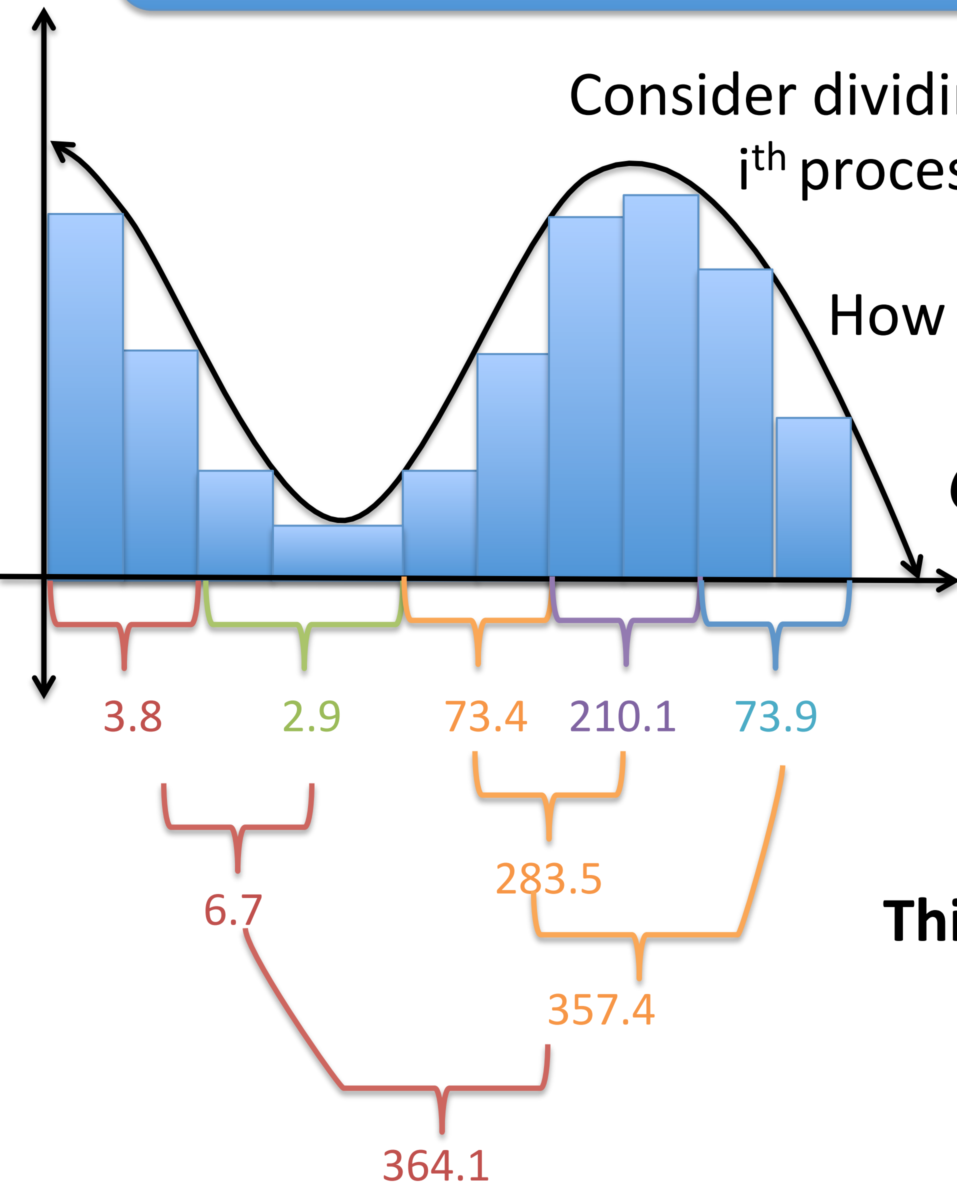
Roll your own:

MPI_Op_create(MPI_User_function* fp,
int <u>commute</u>**,**
MPI_Op *<u>op</u>**);**

```c
void foo_op( void * in, void* inout,
             int * len, MPI_Datatype * dptr){
        //do what you want here
        return;
}
int main(int argc, char* argv[]){
  MPI_Init( &argc, &argc);
  MPI_Op bar_op;
  int this_function_commutes = true;
  MPI_Op_create((MPI_User_function *) foo_op,
                this_function_commutes, &bar_op);
  MPI_Allreduce( .., .., ..,
                bar_op, MPI_COMM_WORLD);
  MPI_Finalize();
  return 1;
}
```

void MPI_User_function(void * invec, void * inoutvec, int * len, MPI_Datatype *datatype)

Problem: Numerical Integration



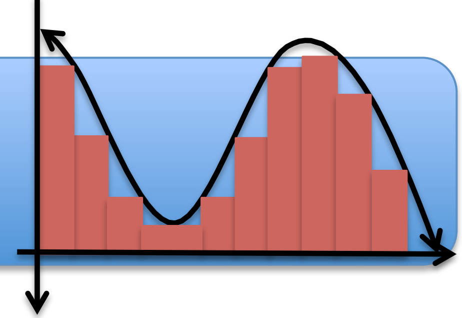
Consider dividing up the domain so that the i^{th} processor gets the i^{th} chunk of rectangles.

How do we combine local sums together?

$O(log_2(p))$ communication rounds

This is difficult to implement correctly.

Reduce



`int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )`

Example:

```
f = A::function();
double local_sum = fast_library::integrate( f, local_a, local_b);
double sum=0; //used only on root processor.
MPI_Reduce( &local_sum,  &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Suppose every processor needs the result:

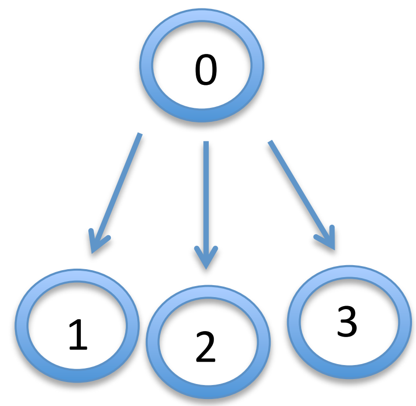
`int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )`

Broadcast

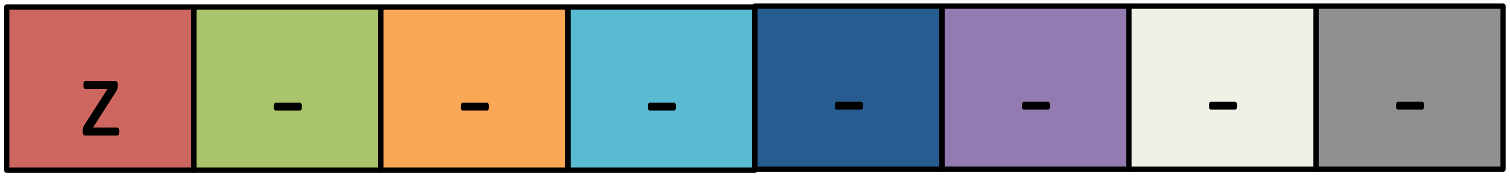
How to send a message from process 0 to all the rest?

This is quite inefficient!

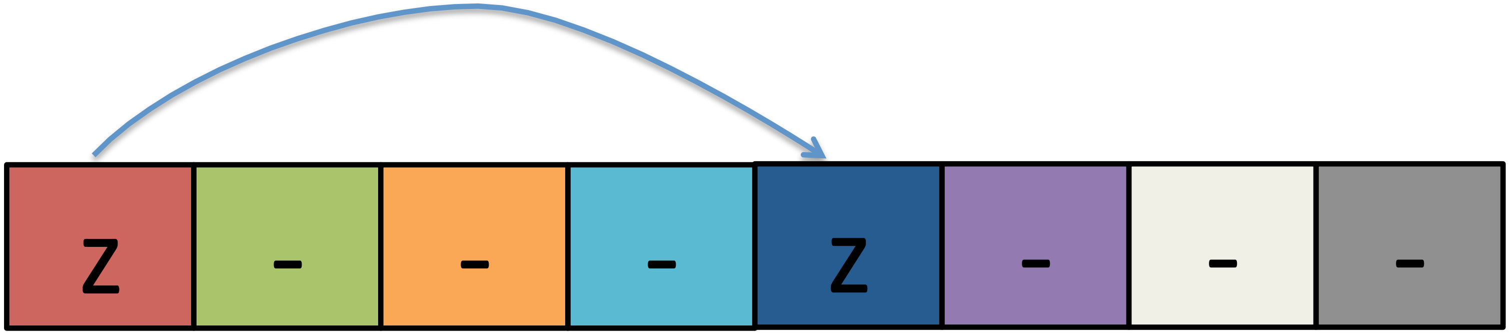
```
void my_bcast(void* data, int count,
              MPI_Datatype datatype, int root,
              MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);

  if (world_rank == root) {
    // If we are the root process, send our data to everyone
    for (std::size_t i = 0; i < world_rank; i++) {
        MPI_Send(data, count, datatype, i, 0, communicator);
    }
    for (std::size_t i = world_rank+1; i < world_size; i++) {
        MPI_Send(data, count, datatype, i, 0, communicator);
    }
  } else {
    //receive the data from the root
    MPI_Recv(data, count, datatype, root, 0, communicator,
             MPI_STATUS_IGNORE);
  }
}
```



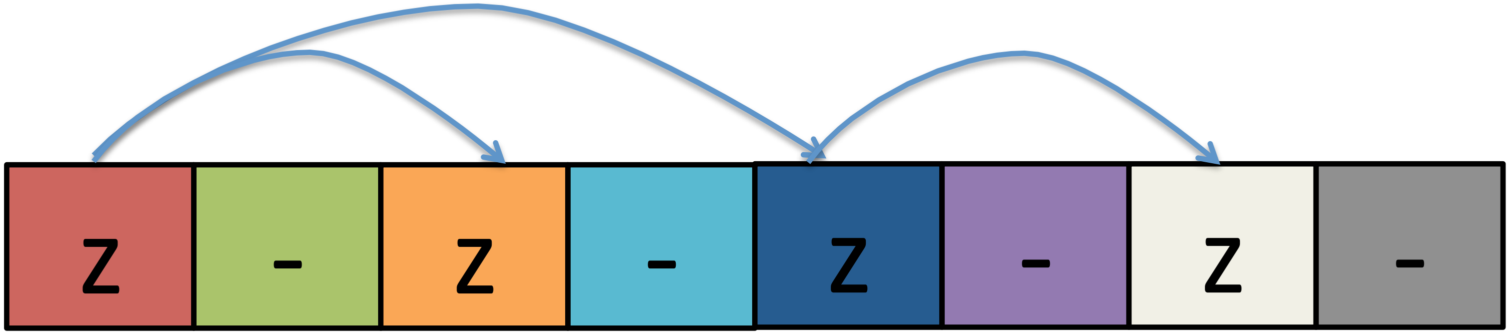
Broadcast



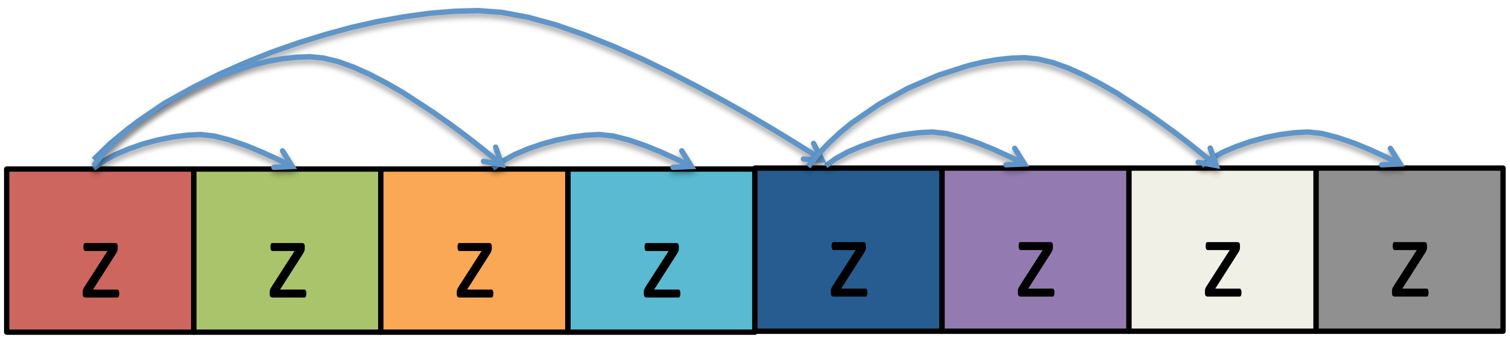
Broadcast



Broadcast



Broadcast



Again, only $O(log_2(p))$ communication rounds

Algorithm sends full copy of each message always.

Broadcast is **dual** to reduce. Reverse all the arrows and get **reduce!**

Again, this is difficult to get right

Broadcast

int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

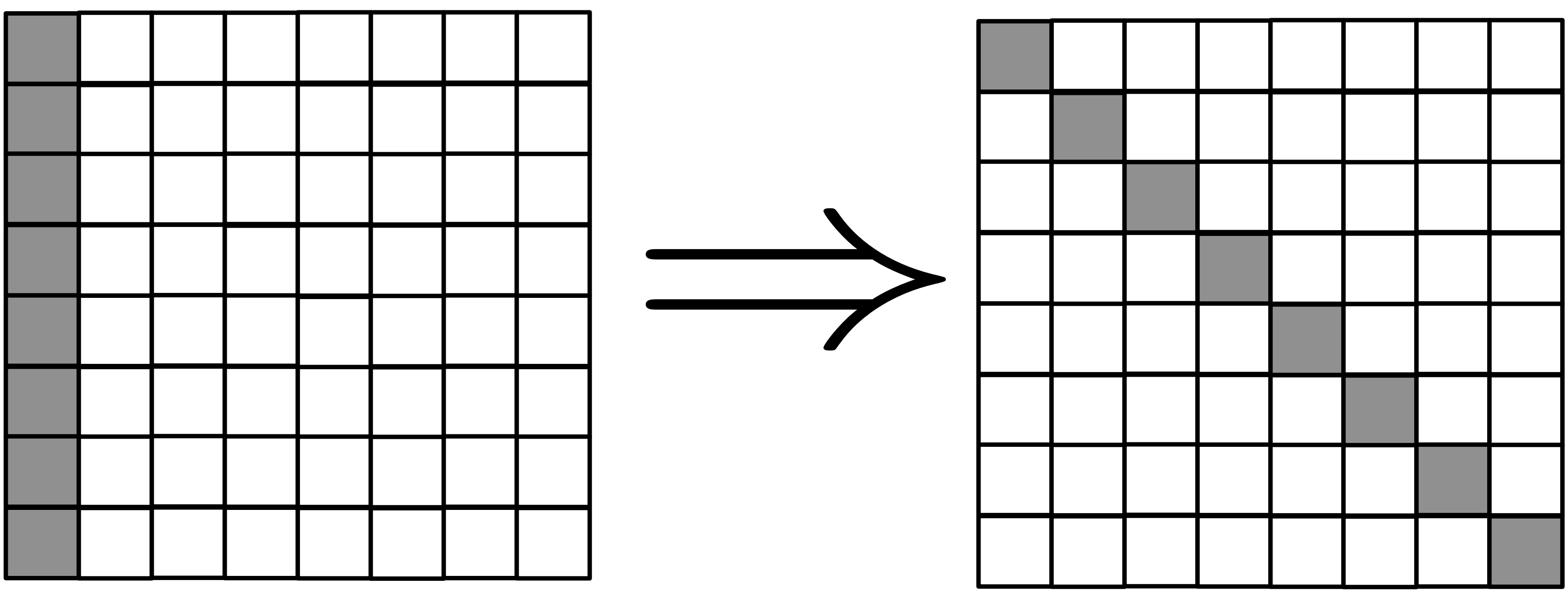
How much better is this algorithm?

```
std::size_t num_trials=10;
std::size_t num_elements = 100000;
//16 processors
for (i = 0; i < num_trials; i++) {
    // Time my_bcast
    // Synchronize before starting timing
    total_my_bcast_time -= MPI_Wtime();
    my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
    total_my_bcast_time += MPI_Wtime();
}
versus
for (i = 0; i < num_trials; i++) {
    total_mpi_bcast_time -= MPI_Wtime();
    MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
    total_mpi_bcast_time += MPI_Wtime();
}
```

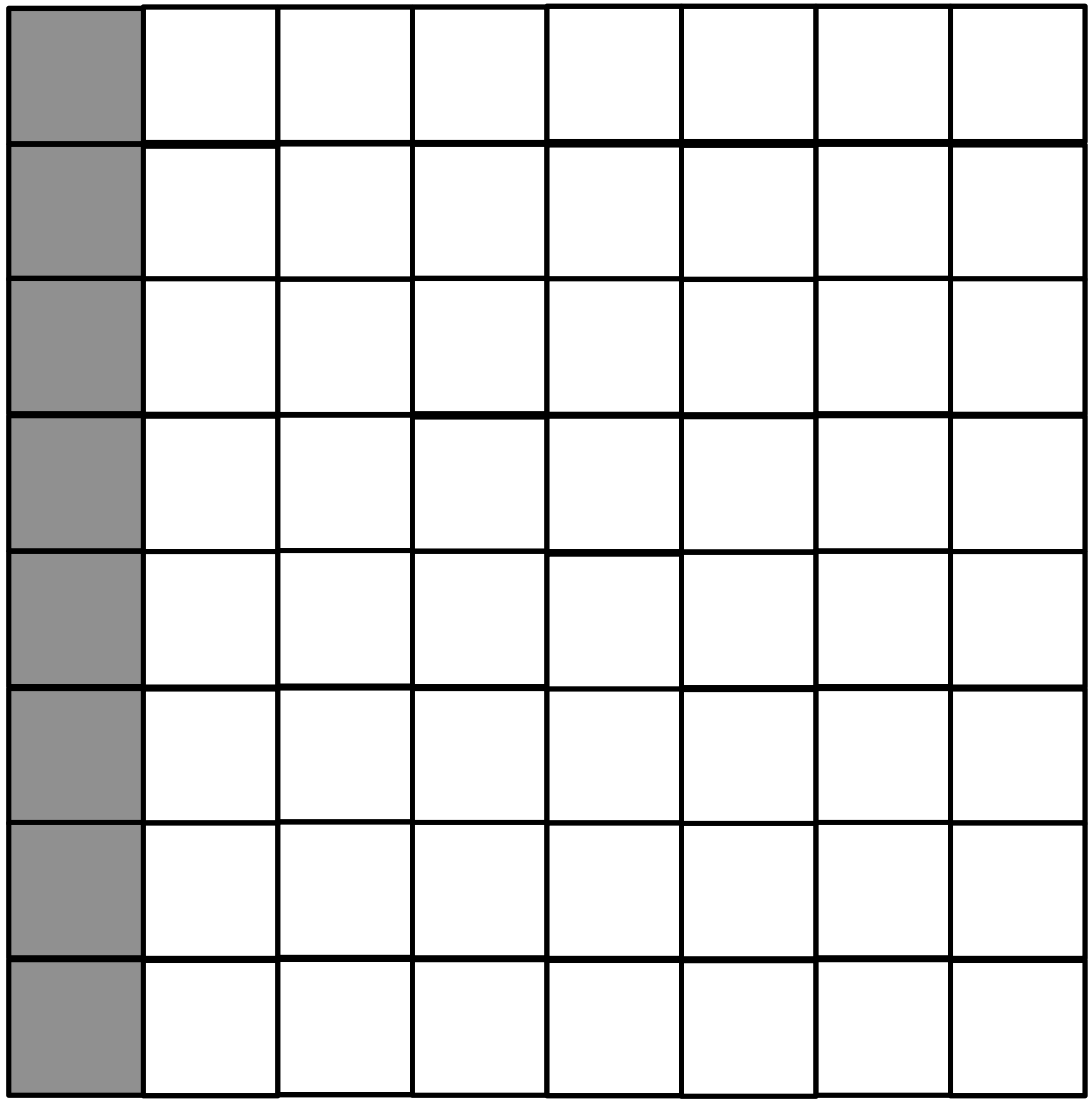
# Processes	my_bcast	MPI_Bcast
2	**0.0344**	**0.0344**
4	0.1025	0.0817
8	0.2385	0.1084
16	**0.5109**	**0.1296**

Scatter (Fox/van-de-Geijn algorithm)

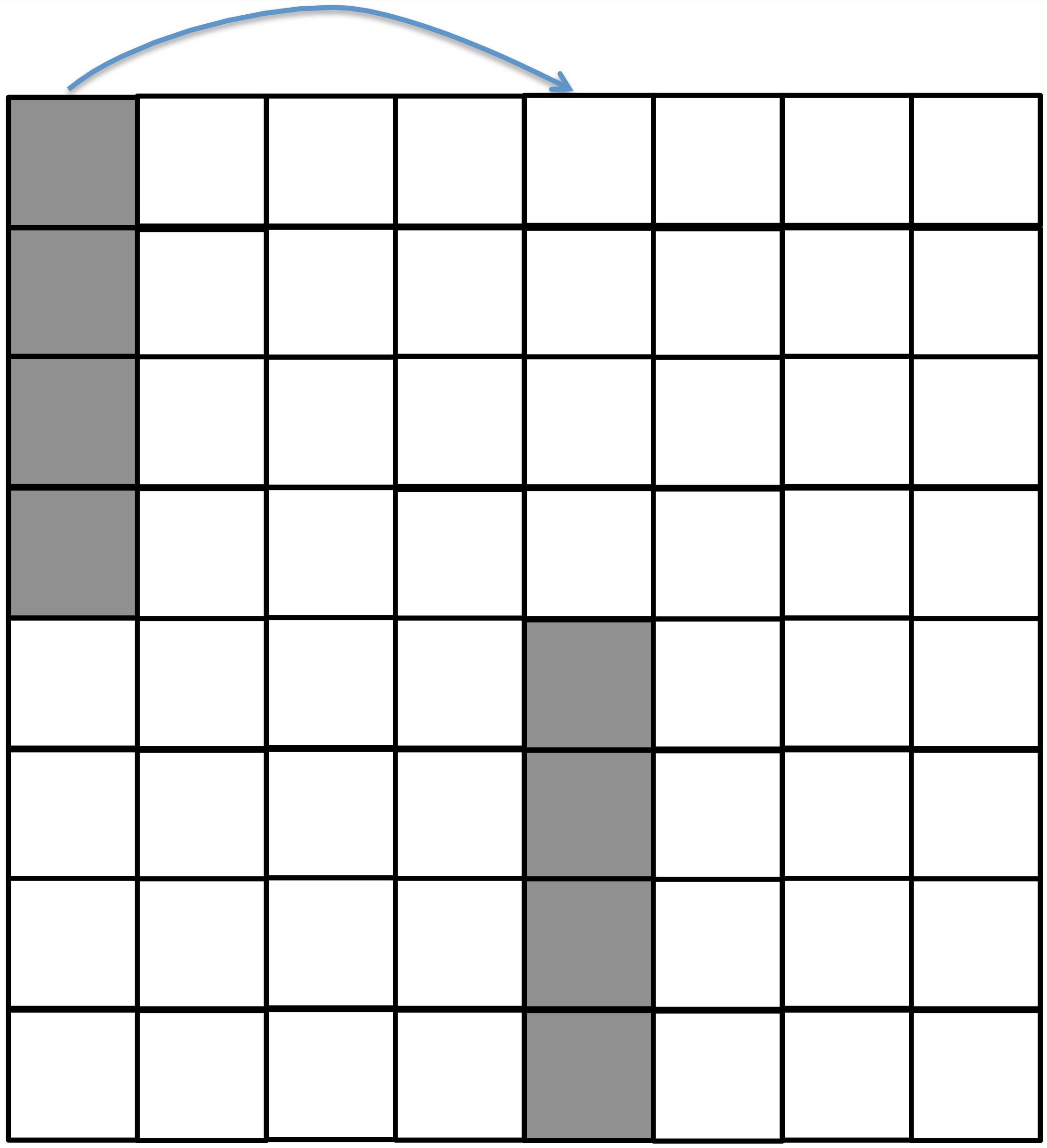
MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)



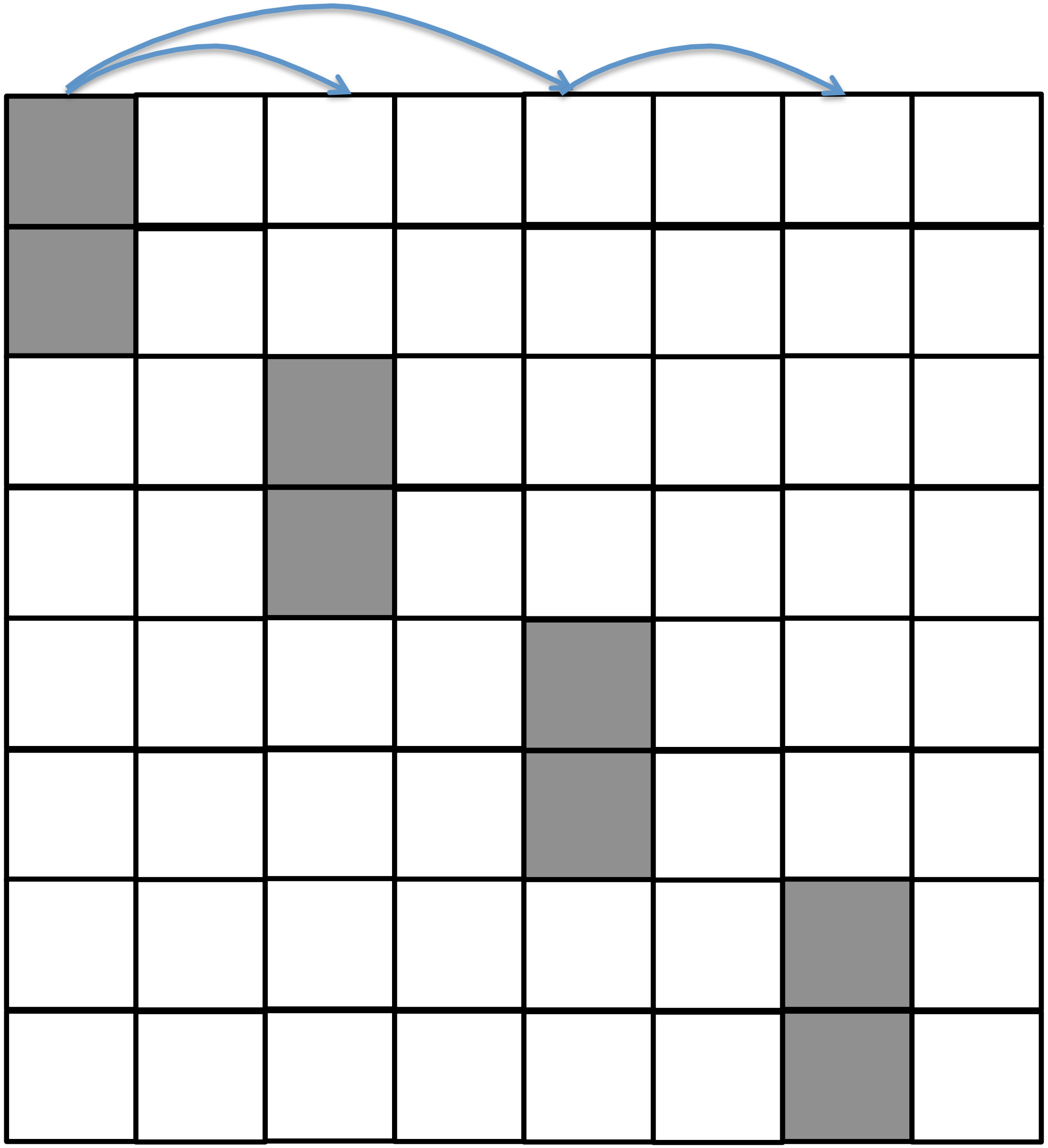
Scatter (Fox/van-de-Geijn algorithm)



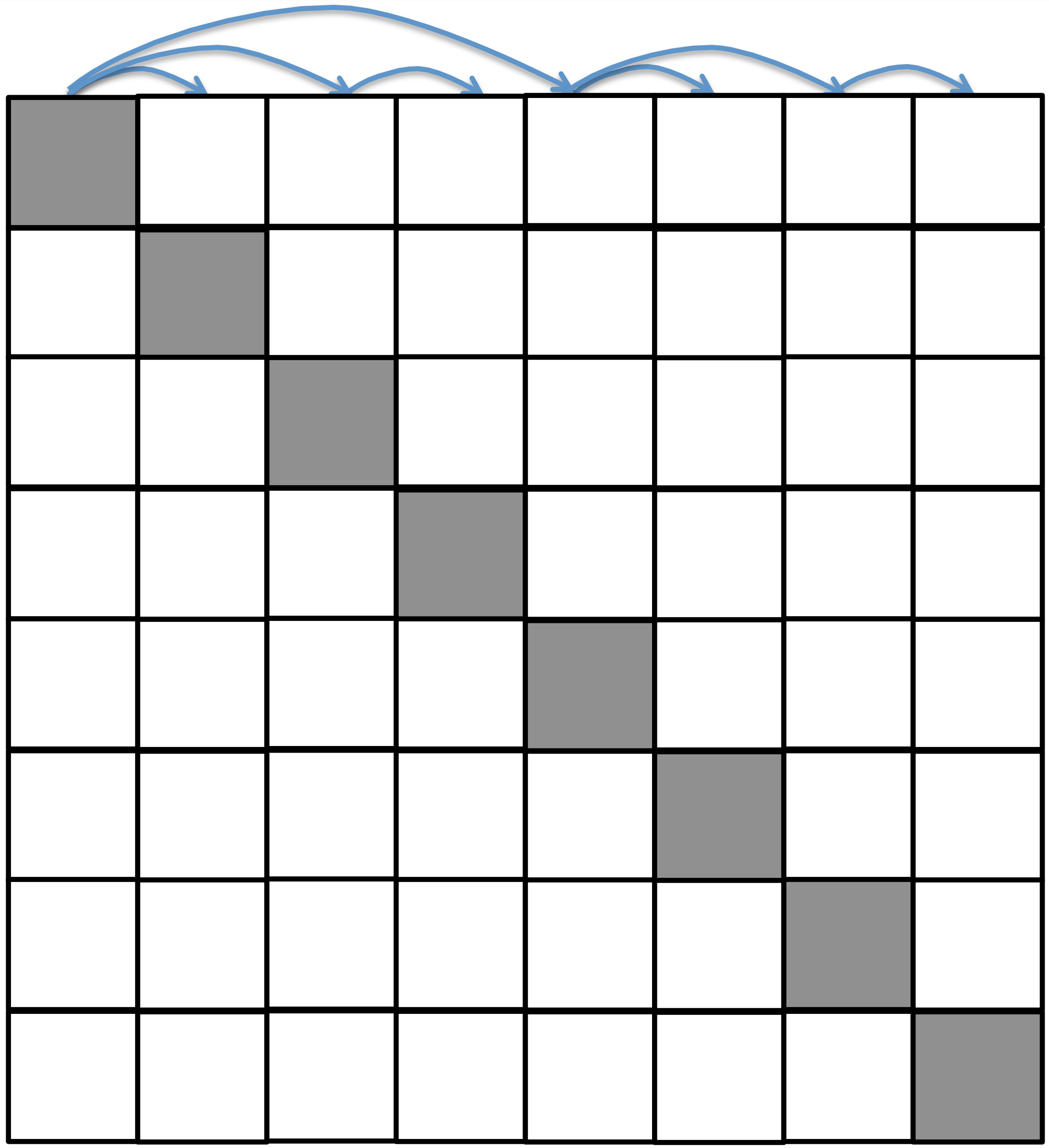
Scatter (Fox/van-de-Geijn algorithm)



Scatter (Fox/van-de-Geijn algorithm)

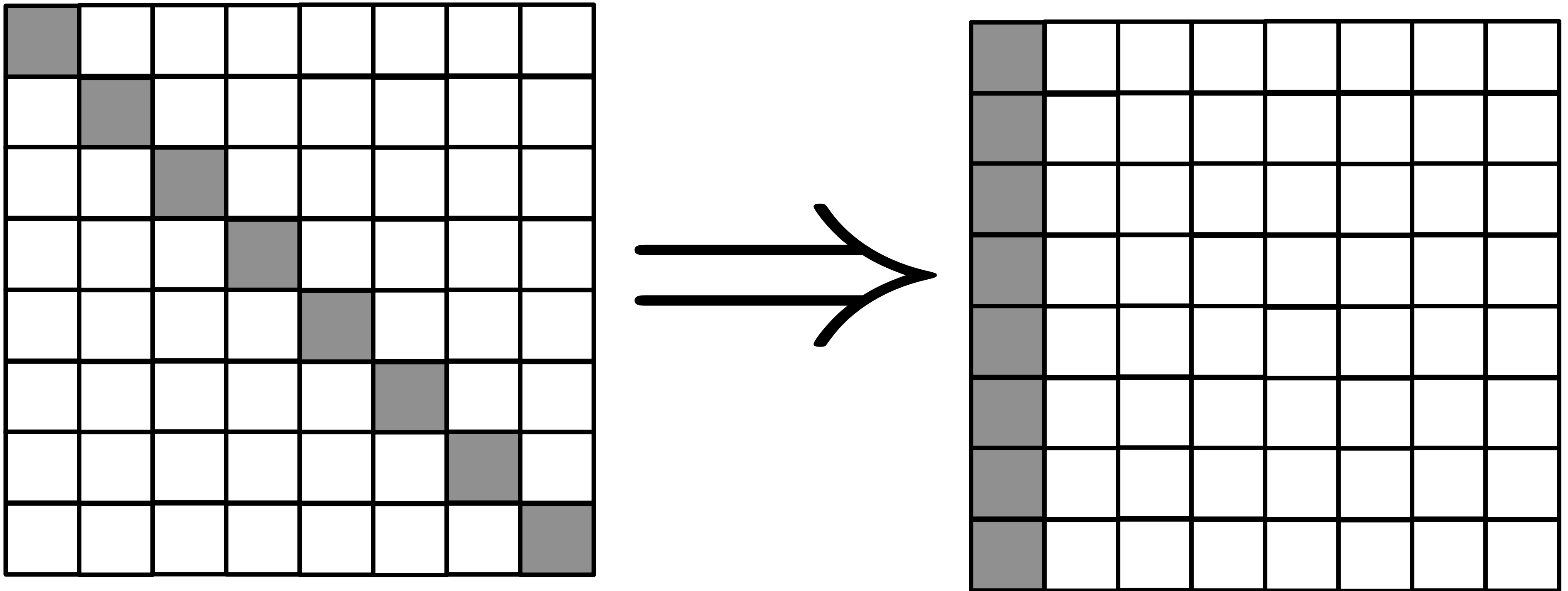


Scatter (Fox/van-de-Geijn algorithm)

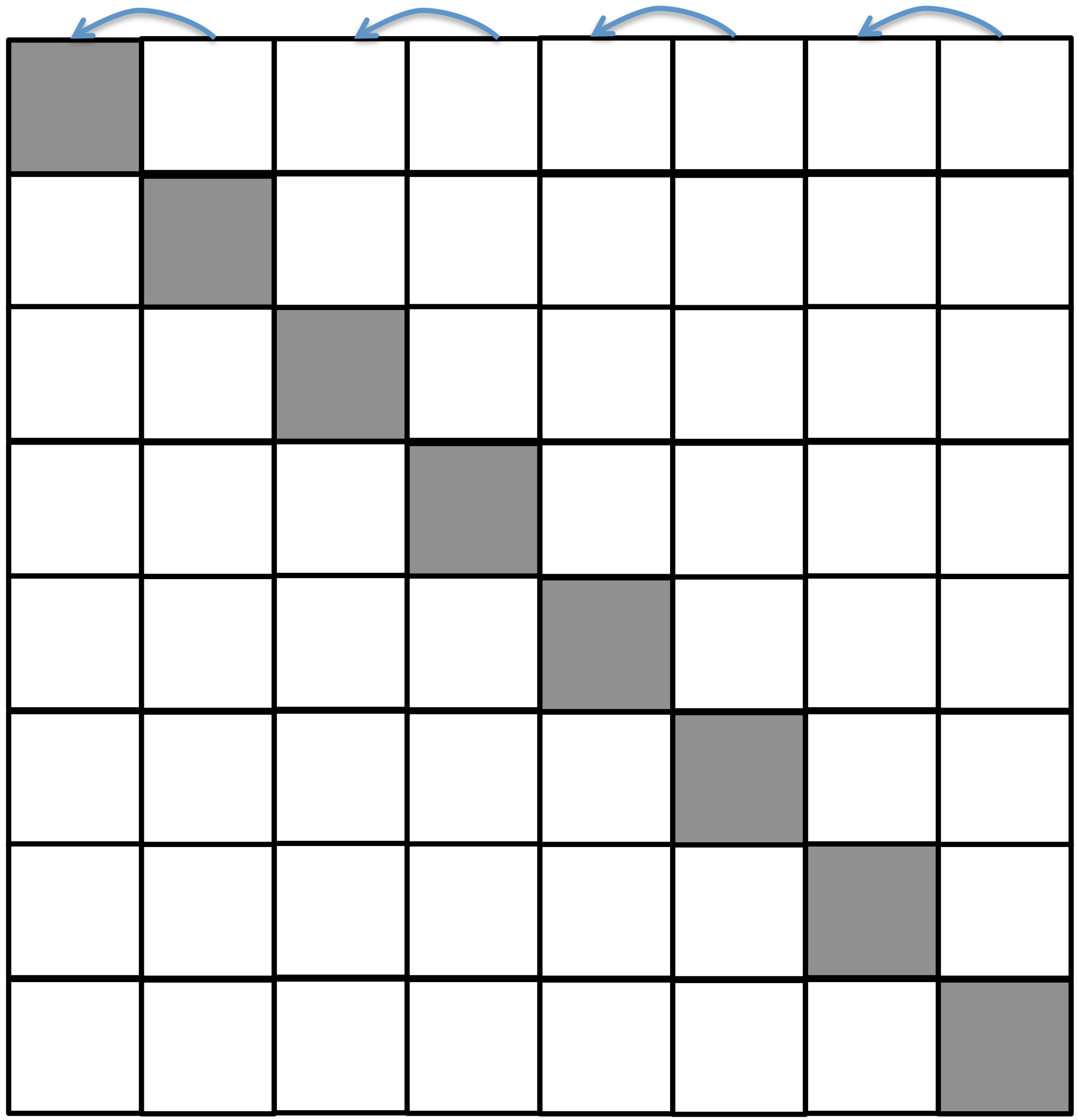


Gather

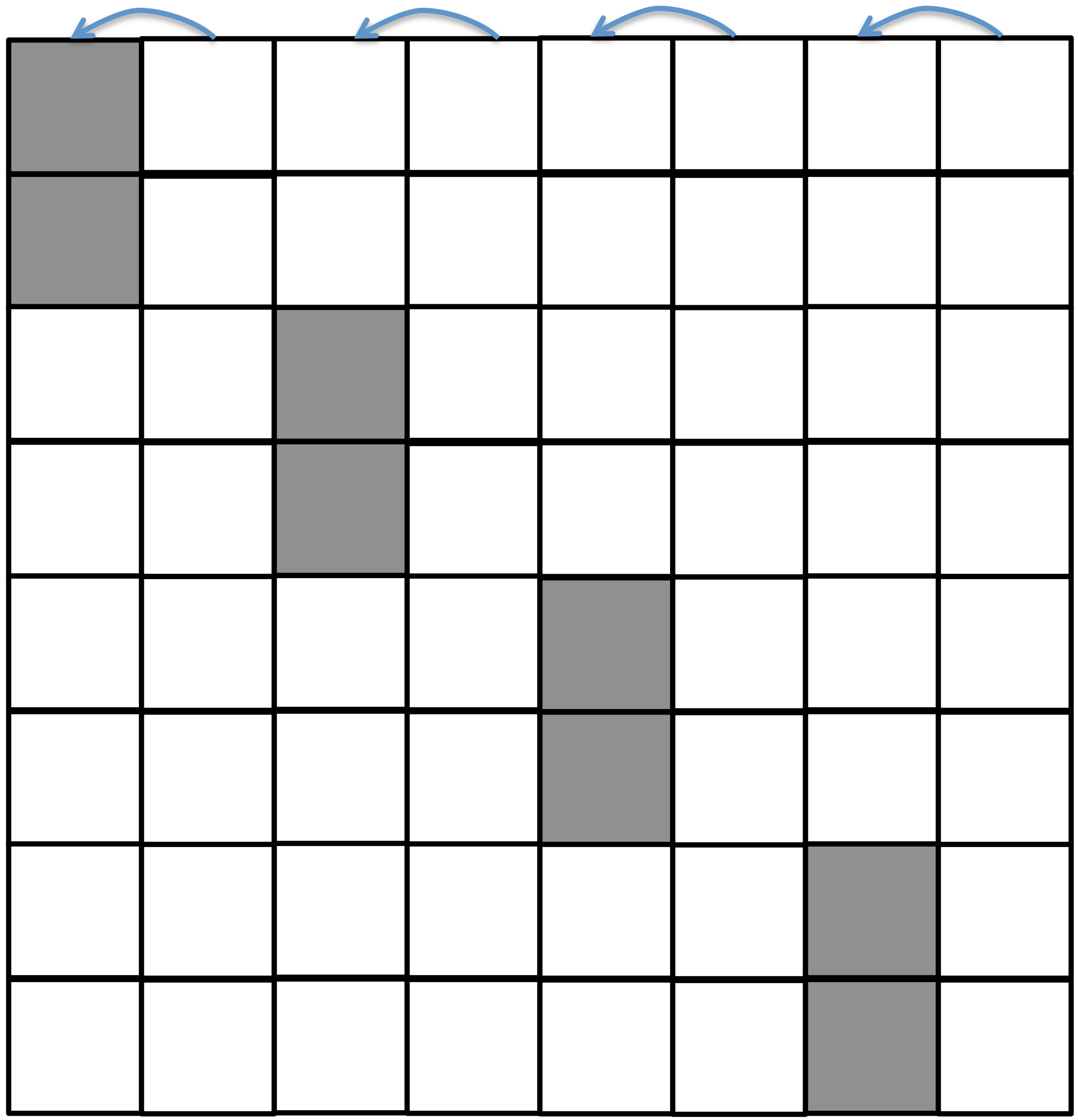
MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)



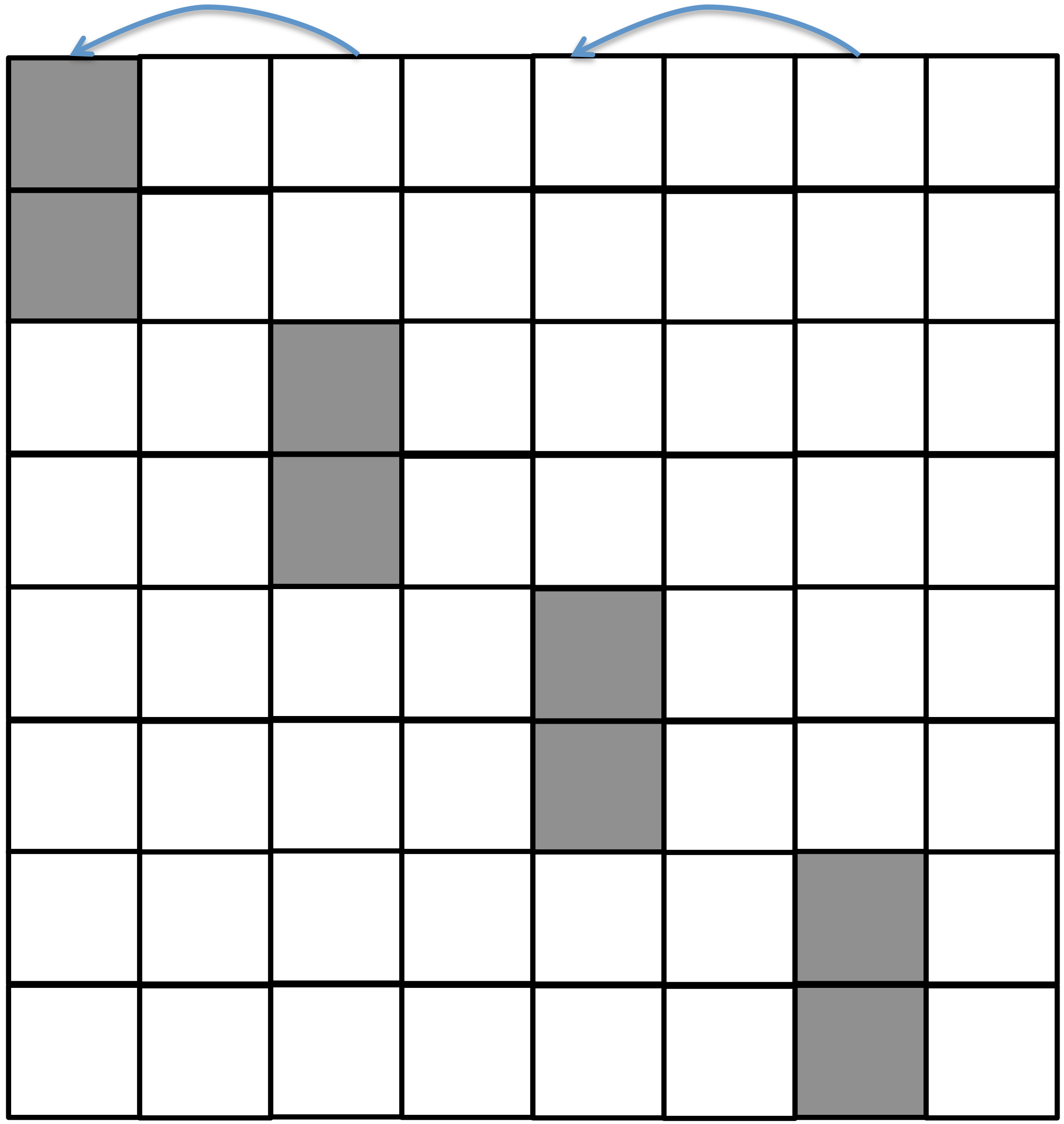
Gather (mhtirogla njieG-ed-nav/xoF)



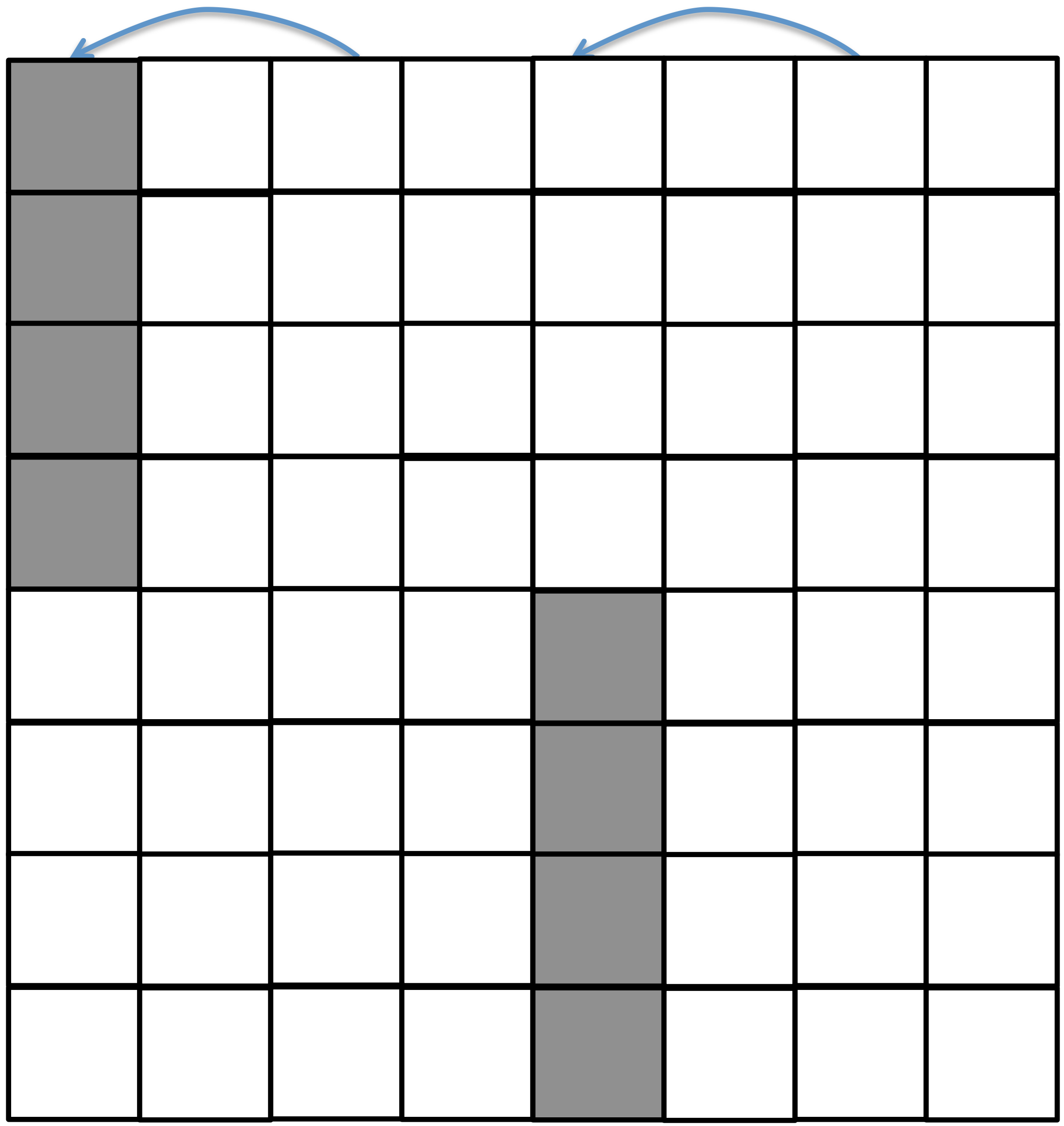
Gather (mhtirogla njieG-ed-nav/xoF)



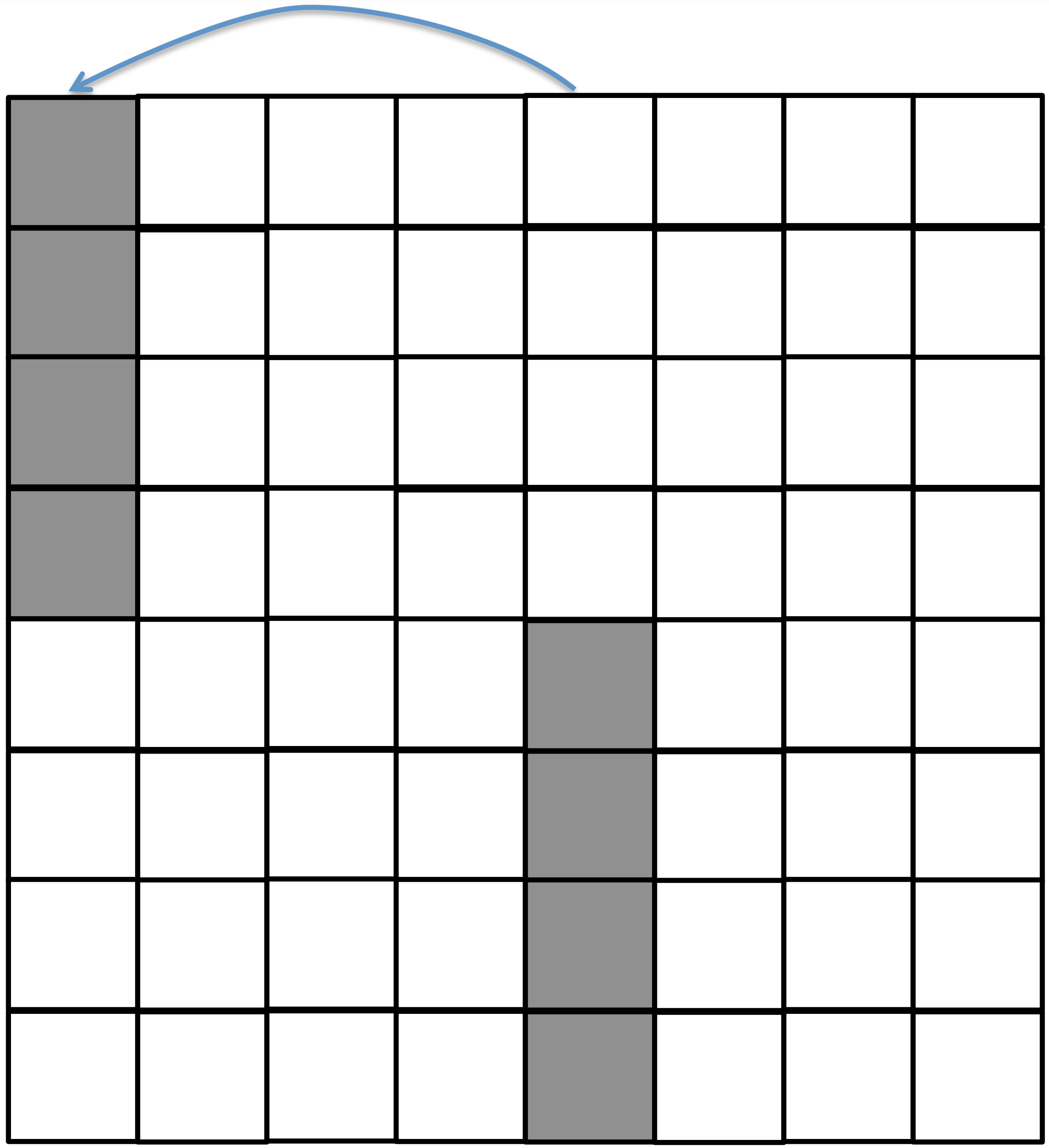
Gather (mhtirogla njieG-ed-nav/xoF)



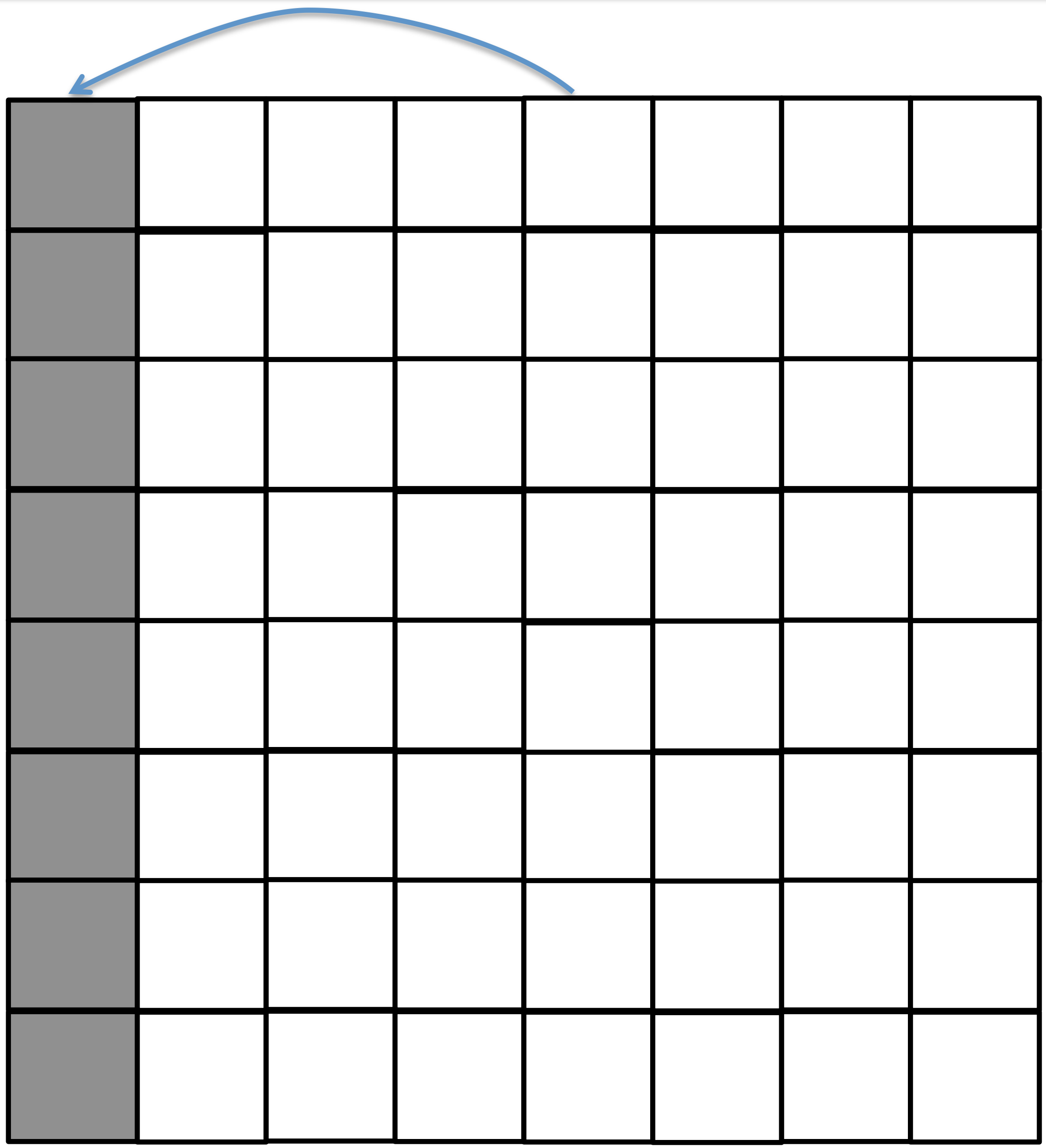
Gather (mhtirogla njieG-ed-nav/xoF)



Gather (mhtirogla njieG-ed-nav/xoF)

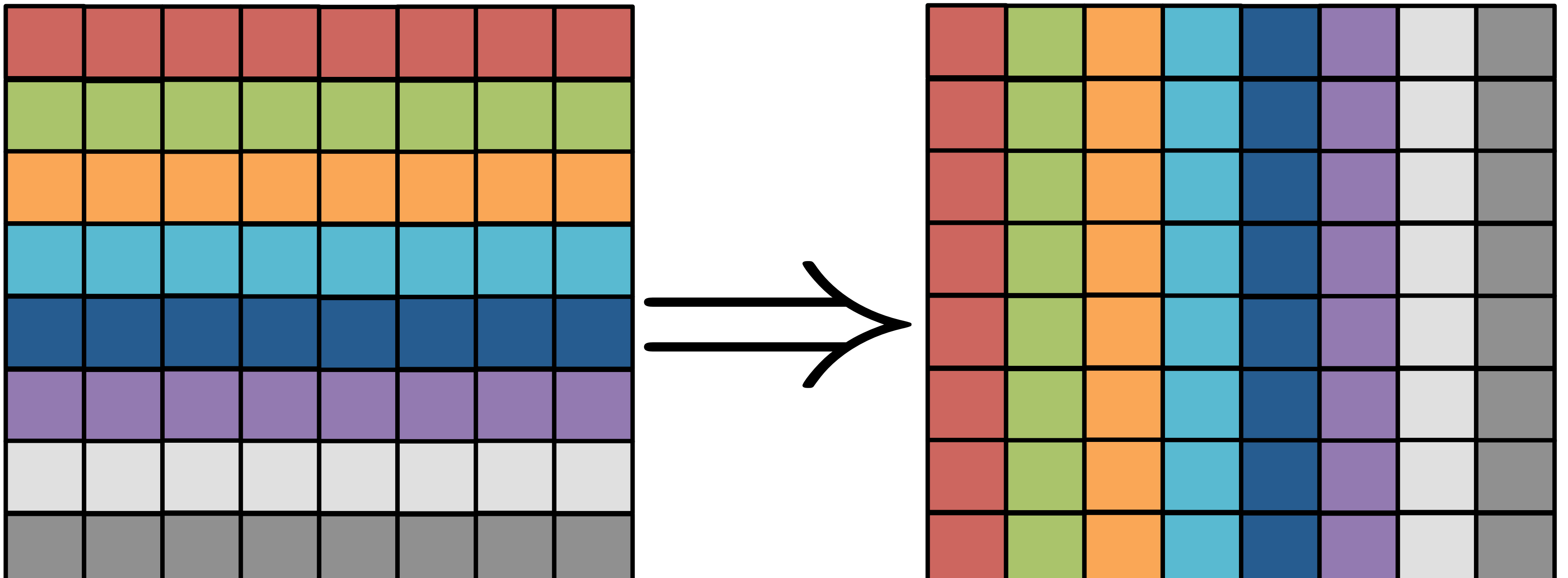


Gather (mhtirogla njieG-ed-nav/xoF)



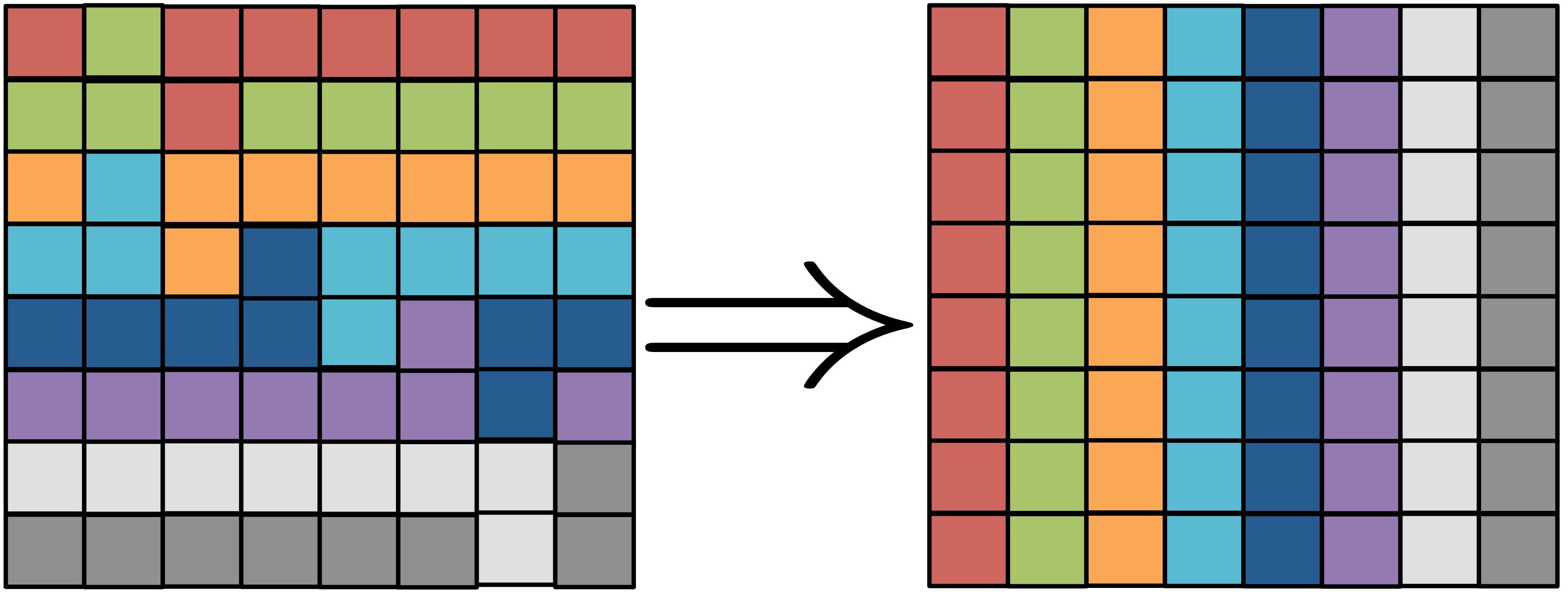
AlltoAll

int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)



AlltoAllv

int MPI_Alltoallv(void* sendbuf, int sendcounts[],
int sdisplsP, MPI_Datatype sendtype,
void* recvbuf, int recvcounts[],
int rdispls[], MPI_Datatype recvtype, MPI_Comm comm)



AlltoAllv

int MPI_Alltoallv(void* sendbuf, int sendcounts[],
int sdisplsP, MPI_Datatype sendtype,
void* recvbuf, int recvcounts[],
int rdispls[], MPI_Datatype recvtype, MPI_Comm comm)

- **sendbuf** Starting address of send buffer.
- **sendcounts** entry i specifies # of elements to send to rank i
- **Sdispls** specifies starting displacements displacements (in units of sendtype)
- **sendtype** Datatype of send buffer elements.

- **recvcounts** entry j specifies the number of elements to receive from rank j
- **Rdispls** Specifies starting displacements displacements (in units of recvtype)
- **Recvtype** Datatype of receive buffer elements.
- **comm** Communicator over which data is to be exchanged.

Scatterv and Gatherv

int MPI_Scatterv(void* sendbuf, int sendcounts[], int displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

These allow us to similarly scatter/gather nonuniform amounts of data.

int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcounts[], int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm)

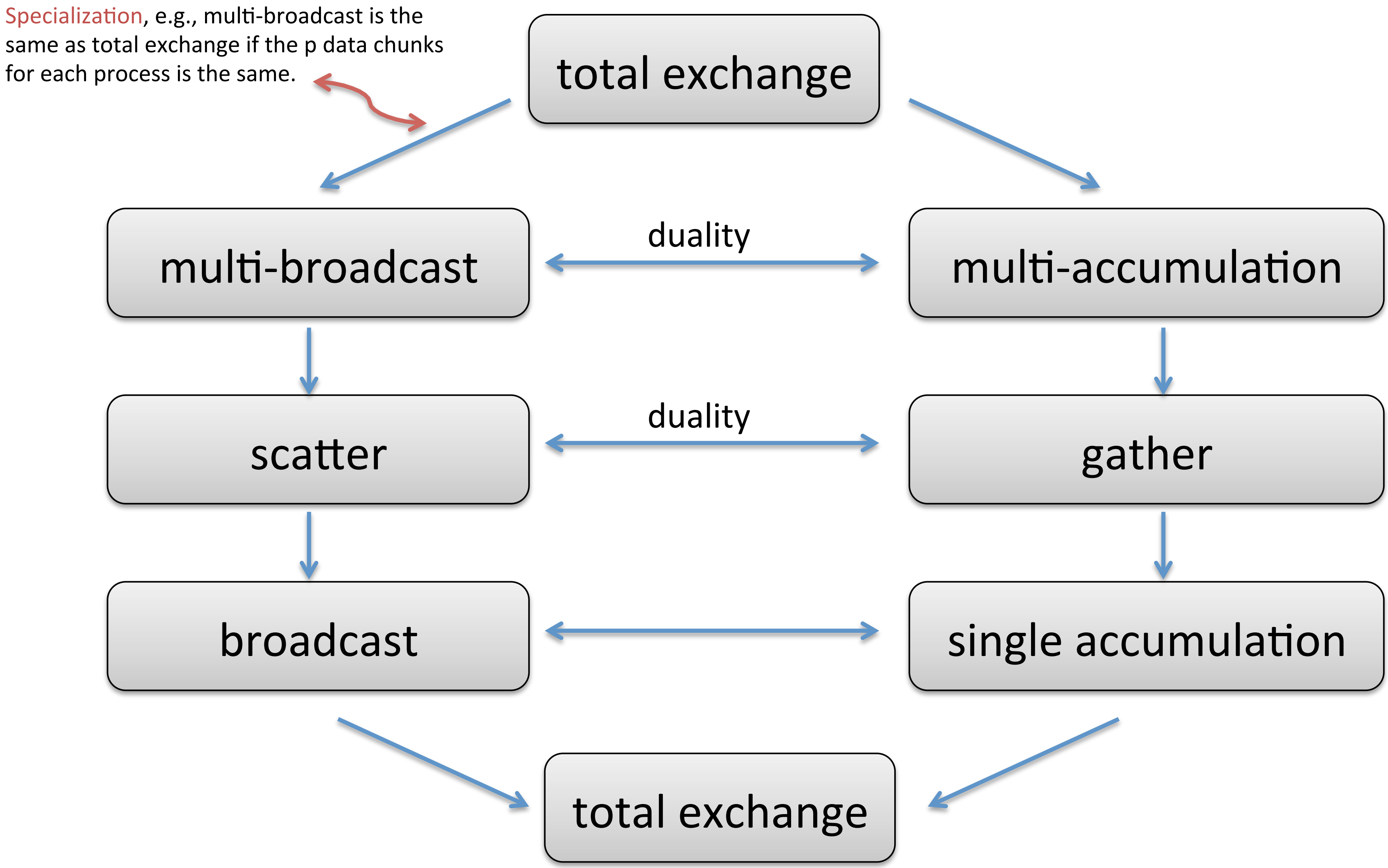
AllScatterv and AllGatherv

int MPI_Allscatterv(void* sendbuf, int sendcounts[], int displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

These allow us to similarly scatter/gather nonuniform amounts of data Between all the machines.

int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcounts[], int displs[], MPI_Datatype recvtype, MPI_Comm comm)

Communication Heirarchy



Coming up

- **Next Monday:** Communicators & Derived Data types
  - Use them with collectives to solve important problems
    - Matrix-Vector Multiplication
    - PDE Solve
- **Penultimate Week:** All about performance
  - Communication modes
  - **Guest Lecture: Rob Schreiber, HP Labs**
- **Final Week:**
  - Existing software using MPI.
  - I'd like volunteers to present on some existing software.
    - Please volunteer or I will volunteer you.

Closing thought experiment

- Suppose you maintain an MPI Library
- Implement AlltoAllv so that each processor would not need to know how much data it was receiving.

Is your solution efficient?